Project Number 732223

D2.3 Dependency Inference and Analysis – Final Report

Version 1.0
29 June 2018
Final

Public Distribution

Centrum Wiskunde & Informatica (CWI)

Project Partners: Athens University of Economics & Business, Bitergia, Castalia Solutions, Centrum Wiskunde & Informatica, Eclipse Foundation Europe, Edge Hill University, FrontEndART, OW2, SOFTEAM, The Open Group, University of L'Aquila, University of York, Unparallel Innovation

Project Partner Contact Information

Athens University of Economics & Business Diomidis Spinellis Patision 76 104-34 Athens Greece Tel: +30 210 820 3621 E-mail: dds@aueb.gr	**Bitergia** José Manrique Lopez de la Fuente Calle Navarra 5, 4D 28921 Alcorcón Madrid Spain Tel: +34 6 999 279 58 E-mail: jsmanrique@bitergia.com
Castalia Solutions Boris Baldassari 10 Rue de Penthièvre 75008 Paris France Tel: +33 6 48 03 82 89 E-mail: boris.baldassari@castalia.solutions	**Centrum Wiskunde & Informatica** Jurgen J. Vinju Science Park 123 1098 XG Amsterdam Netherlands Tel: +31 20 592 4102 E-mail: jurgen.vinju@cwi.nl
Eclipse Foundation Europe Philippe Krief Annastrasse 46 64673 Zwingenberg Germany Tel: +33 62 101 0681 E-mail: philippe.krief@eclipse.org	**Edge Hill University** Yannis Korkontzelos St Helens Road Ormskirk L39 4QP United Kingdom Tel: +44 1695 654393 E-mail: yannis.korkontzelos@edgehill.ac.uk
FrontEndART Rudolf Ferenc Zászló u. 3 I./5 H-6722 Szeged Hungary Tel: +36 62 319 372 E-mail: ferenc@frontendart.com	**OW2 Consortium** Cedric Thomas 114 Boulevard Haussmann 75008 Paris France Tel: +33 6 45 81 62 02 E-mail: cedric.thomas@ow2.org
SOFTEAM Alessandra Bagnato 21 Avenue Victor Hugo 75016 Paris France Tel: +33 1 30 12 16 60 E-mail: alessandra.bagnato@softeam.fr	**The Open Group** Scott Hansen Rond Point Schuman 6, 5th Floor 1040 Brussels Belgium Tel: +32 2 675 1136 E-mail: s.hansen@opengroup.org
University of L′Aquila Davide Di Ruscio Piazza Vincenzo Rivera 1 67100 L′Aquila Italy Tel: +39 0862 433735 E-mail: davide.diruscio@univaq.it	**University of York** Dimitris Kolovos Deramore Lane York YO10 5GH United Kingdom Tel: +44 1904 325167 E-mail: dimitris.kolovos@york.ac.uk
Unparallel Innovation Bruno Almeida Rua das Lendas Algarvias, Lote 123 8500-794 Portimão Portugal Tel: +351 282 485052 E-mail: bruno.almeida@unparallel.pt	

Document Control

Version	Status	Date
0.1	First draft	2 June 2018
0.6	First internal release	22 June 2018
1.0	Final release	29 June 2018

Table of Contents

Executive Summary

This document reports on the results obtained for **Task 2.1** and **Task 2.2**:

Task 2.1: Inference of project build configuration This task will analyse source code project build configurations for (at least) Maven and Eclipse based projects to extract precisely which other projects or libraries a project depends on. Where possible dependency analysis will be extended with findings derived from operating system-level package managers. Next to analysing project meta-data, this task also entails extracting definitions from previously deployed open-source projects and their meta-data on the Maven grand central and the most recent P2 repository by Eclipse;

Task 2.2: Modelling framework semantics This task will use the acquired meta-data on project dependency in combination with in-depth analysis of the source code of open-source projects to produce actionable information about factual dependencies. The produced information will be more accurate (avoiding both junk and confusion) than manual documentation. The resulting analyses should be amenable to bespoke extensions.

In this deliverable, we present a set of methods and tools for extracting actionable knowledge from meta-data specification of dependencies in the OSGi[1] and Apache Maven[2] frameworks. As Maven has already been the subject of extensive research, we put a particular emphasis on the OSGi framework and its use in the P2 repositories of Eclipse. P2 is a provisioning platform where software solutions based on Eclipse and Equinox (Eclipse implementation of OSGi) are managed. Besides, OSGi is heavily used in the Eclipse use case presented in “Use Case 6: Knowledge Extraction from Eclipse Forge Projects” (cf. **D1.1 – Project Requirements**). To better understand how dependencies are currently managed by developers, and how CROSSMINER should help in this regard, we conduct an empirical study of the use of OSGi in Eclipse.

We first conduct a systematic literature review that reveals a set of best practices commonly advocated by experts and practitioners of the OSGi framework. We use these best practices to drive the definition of bespoke recommendations in the project. Then, we present an analysis tool written in Rascal that extracts a set of metrics from OSGi bundles in order to check whether these best practices are followed in practice. Using a large corpus of OSGi bundles (the set of core plug-ins of Eclipse 4.6 Neon), we show whether developers follow these best practices and discuss their impact on OSGi-based systems.

The goal of this deliverable is to lay an empirical foundation to build further upon when producing actual feedback and suggestion components for the CROSSMINER platform. It reports on the necessary steps towards accurate analysis of dependencies within the CROSSMINER project and IDE, and meaningful recommendations for the developer in the IDE.

As the semantics of Apache Maven and OSGi largely differ on many aspects as it can be inferred from the official documentation, we do not attempt to define a unified analysis framework or a set of common metrics for both of these.

Part of the material presented here has been published and presented at the *15th International Conference on Mining Software Repositories* (MSR’18) [26]. The supporting paper can be accessed freely at the following location: `https://hal.archives-ouvertes.fr/hal-01740131/document`.

We refer the reader to the companion deliverable **D2.4 – Dependency Inference Components** for a description of the concrete software artefacts we developed to support **Task 2.1** and **Task 2.2** and the way they are integrated within the overall CROSSMINER platform.

[1]`https://www.osgi.org/`
[2]`https://maven.apache.org/`

This final report includes the following content:

- In Part I, we give some background on the OSGi framework and conduct a systematic review to extract a set of best practices commonly advocated by OSGi experts and practitioners. Then, we specify, apply, and evaluate an analysis tool on a large corpus of projects originating from the *Eclipse Europe Foundation* use case partner to discover whether (i) these best practices are being followed by Eclipse developers and (ii) what is their impact on OSGi-based systems;
- In Part II, we present some background on the Apache Maven framework and present a methodology for extracting actionable knowledge from an analysis of Maven projects meta-data regarding dependencies management;
- In Part III, we relate the work presented here to other work packages of the CROSSMINER project and review the requirements originating from use case partners that we address.

Part I

An Empirical Evaluation of OSGi Dependencies Best Practices

1 Introduction

The time-honoured principle of separation of concerns entails splitting the development of complex systems into multiple components interacting through well-defined interfaces. This way, the development of a system can be broken down into multiple, smaller parts that can be implemented and tested independently. This also fosters reuse by allowing software components to be reused from one system to the other, or even to be substituted by one another provided that they satisfy the appropriate interface expected by a client. Three crucial aspects [29] of successful separation of concerns are module interfaces, module dependencies, and information hiding—a module's interface hides any number of different functionalities, possibly depending on other modules transitively.

Historically, the Java programming language did not offer any built-in support for the definition of versioned modules with explicit dependency management [36]. This led to the emergence of OSGi, a module system and service framework for Java standardized by the OSGi Alliance organization [37]. Initially, one of the primary goals of OSGi was to fill the lack of proper support for modular development in the Java ecosystem (popularly known as the "JAR hell") OSGi rapidly gained popularity and, as of today, numerous popular software of the Java ecosystem, including IDEs (e.g., Eclipse, IntelliJ), application servers (e.g., JBoss, GlassFish), and application frameworks (e.g., Spring) rely internally on the modularity capabilities provided by OSGi.

Just like any other technology, it may be hard for newcomers to grasp the complexity of OSGi. The OSGi specification describes several distinct mechanisms to declare dependencies, each with different resolution and wiring policies (i.e., how to match bundle requirements against bundle capabilities when dependencies are specified). Should dependencies be declared at the package level or the component level? Can the content of a package be split amongst several components or should it be localized in a single one? These are questions that naturally arise when attempting to modularize Java applications with OSGi. There is little tool support to help writing the meta-data files that wire the components together, and so modularity design decisions are mostly made by the developers themselves. The quality of this meta-data influences the modularity aspects of OSGi systems. The reason is that OSGi's configurable semantics directly influences all the aforementioned key aspects of modularity: the definition of module interfaces, what a dependency means (wiring), and information hiding (e.g., transitive dependencies).

A conventional approach to try and avoid such issues is the application of so-called "best practices" advised by experts in the field. To the best of our knowledge, the assumptions underlying this advice have not been investigated before: are they indeed relevant and do they have a positive effect on OSGi-based systems? Our research questions are:

Q1 What OSGi best practices are advised?

Q2 Are OSGi best practices being followed?

Q3 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?

To begin answering these questions, this first part of the deliverable makes the following contributions:

- A systematic review of best practices for dependency management in OSGi emerging from either the OSGi Alliance itself or OSGi-endorsed partners; we identify 11 best practices and detail the rationale behind them (**Q1**);
- An analysis of the bytecode and meta-data of a representative corpus of OSGi bundles (1,124 core plug-ins of the Eclipse IDE) to determine whether best practices are being followed (**Q2**), and what is their impact (**Q3**); this corpus emerges from the *Eclipse Europe Foundation* use case in the CROSSMINER project.

Our results show that:

- *Best practices are not widely followed in practice*. For instance, half of the bundles we analyse specify dependencies at the bundle level rather than at the package level—despite the fact that best practices encourage to declare dependencies at the package level;
- *The lack of consideration for best practices does not significantly impact the performance of OSGi-based systems*. Strictly following the suggested best practices reduces classpath size of individual bundles by up to 23% and results in up to ±13% impact on performance at bundle resolution time.

The remaining of this part is structured as follows:

- In Section 2, we introduce background notions on the OSGi framework itself;
- In Section 3, we detail the methodology of the systematic literature review from which we extract a set of best practices related to dependencies management;
- In Section 4, we present a set of metrics that provide factual information related to the use of these best practices;
- In Section 5, we present our analysis tool for OSGi implemented in Rascal;
- In Section 6, we evaluate whether best practices are followed and what is their impact on a representative corpus of OSGi bundles extracted from the Eclipse IDE;
- In Section 7, we give some concluding remarks on our analysis of OSGi.

2 Background: the OSGi Framework

OSGi is a module system and service framework for the Java programming language standardized by the OSGi Alliance organization [37], which aims at filling the lack of support for modular development with explicit dependencies in the Java ecosystem (aka. the "JAR hell"). Some of the ideas that emerged in OSGi were later incorporated in the Java standard itself, e.g., as part of the module system released with Java 9. In OSGi, the primary unit of modularization is a *bundle*. A bundle is a cohesive set of Java packages and classes (and possibly other arbitrary resources) that together provide some meaningful functionality to other bundles. A bundle is typically deployed in the form of a Java archive file (JAR) that embeds a *Manifest file* describing its content, its meta-data (e.g., version, platform requirements, execution environment), and its dependencies towards other bundles. The OSGi framework itself is responsible for managing the life cycle of bundles (e.g., installation, startup, pausing), possibly remotely. As of today, several certified implementations of the OSGi specification have been defined, including Eclipse Equinox[3] and Apache Felix[4] to name but a few.

OSGi is a mature framework that comprises many aspects ranging from module definition and service discovery to life cycle and security management [37]. In this document, we focus specifically on its support for dependencies management. We describe in Section 2.1 the purpose and syntax of Manifest files, and then detail in Section 2.2 how bundles declare their dependencies and how they are wired by the framework.

2.1 The Manifest File

Every bundle contains a meta-data file located in `META-INF/MANIFEST.MF`. This file contains a list of standardized key-value pairs (known as *headers*) that are interpreted by the framework to ensure all requirements of the bundle are met. Listing 1 depicts an idiomatic Manifest file for an imaginary bundle named `Dummy`.

```
Bundle-ManifestVersion: 2
Bundle-Name: Dummy
Bundle-SymbolicName: a.dummy
Bundle-Version: 0.2.1.build-21
Bundle-RequiredExecutionEnvironment: JavaSE-1.8
Export-Package: a.dummy.p1,
 a.dummy.p2;version="0.2.0"
Import-Package: b.p1;version="[1.11,1.13]",
 c.p1
Require-Bundle: d.bundle;bundle-version:="3.4.1",
 e.bundle;resolution:=optional
```

Listing 1: An idiomatic `MANIFEST.MF` file.

In this simple example, the Manifest file declares the bundle `a.Dummy` in its version `0.2.1.build-21`. It requires the execution environment `JavaSE-1.8`. The main purpose of this header is to announce what should be available to the bundle in the standard `java.*` namespace, as the exact content may vary according to the version and the implementer of the Java virtual machine on which the framework runs. The Manifest file specifies that the bundle exports the `a.dummy.p1` package, and the `a.dummy.p2` package in version `0.2.0`. These packages form the public interface of the bundle—its API. Next, the Manifest file specifies that the bundle requires the package `b.p1` in version `1.11` to `1.13` (inclusive) and the package `c.p1`. Finally, the Manifest declares a

[3] https://www.eclipse.org/equinox/
[4] https://felix.apache.org/

Table 1: OSGi dependencies headers [37].

Header	Description
`Bundle-SymbolicName`	"[...] *together with a version must identify a unique bundle*"
`Bundle-Version`	"[...] *specifies the version of this bundle*"
`DynamicImport-Package`	"[...] *contains a comma-separated list of package names that should be dynamically imported when needed*"
`Export-Package`	"[...] *contains a declaration of exported packages*"
`Import-Package`	"[...] *declares the imported packages for this bundle*"
`Require-Bundle`	"[...] *specifies that all exported packages from another bundle must be imported, effectively requiring the public interface of another bundle*"

dependency towards the bundle `d.bundle` in version `3.4.1` and an optional dependency towards the bundle `e.bundle`. We dive into greater details of the semantics of these headers and attributes in the next section.

It is important to note that the Manifest file is typically written by the bundle's developer herself and has to co-evolve with its implementation. Therefore, discrepancies between what is declared in the Manifest and what is actually required by the bundle at the source or bytecode level may arise. Although some tools provide assistance to the developers (for instance using bytecode analysis techniques on bundles to automatically infer the appropriate dependencies), getting the Manifest right remains a tedious and error-prone task [34].

2.2 OSGi Dependencies Management

The OSGi specification declares 28 Manifest headers that relate to versioning, internationalization, dependencies, capabilities, etc. Amongst them, six are of particular interest regarding dependencies management. They are listed in Table 1. The OSGi specification prescribes two distinct mechanisms for declaring dependencies: at the package level, or at the bundle level. In the former case, it is the responsibility of the framework to figure out which bundle provides the required package—multiple bundles can export the same package in the same version. Conversely, the latter explicitly creates a strong dependency link between the two bundles.

The **`Import-Package`** header consists of a list of comma-separated packages the declaring bundle depends on. Each package in the list accepts an optional list of attributes that affects the way packages are resolved. The *`resolution`* attribute accepts the values `mandatory` (default) and `optional`, which indicate, respectively, that the package must be resolved for the bundle to load, or that the package is optional and will not affect the resolution of the requiring bundle. The *`version`* attribute restricts the resolution on a given version range, as shown in Listing 1.

When it requires another bundle through the **`Require-Bundle`** header, a bundle imports not only a single package but the whole public interface of another bundle, i.e., the set of its exported packages. As the **`Require-Bundle`** header requires to declare the symbolic name of another bundle explicitly, this creates a strong dependency link between both. Thus, not only does this header operate on a coarse-grained unit of modularization, but it also tightly couples the components together.

For a bundle to be successfully resolved, all the packages it imports must be exported (**`Export-Package`**) by some other bundle known to the framework, with their versions matching. Similarly, all the bundles it requires must be known to the framework, with their versions matching. This wiring process is carried out automatically by the framework as the bundles are loaded.

3 OSGi Best Practices

The OSGi specification covers numerous topics in depth and it can be hard for developers to infer idiomatic uses and good practices. Should dependencies be declared at the package or the bundle level? Can the content of a package be split amongst several bundles or should it be localized in a single one? These are questions that naturally arise when attempting to modularize Java applications with OSGi. Although all usages are valid according to the specification, OSGi experts tend to recommend or discourage some of them.

In this section, we intend to identify a set of best practices in the use of OSGi. We present the systematic review methodology we employ (Section 3.1) and describe the 11 OSGi best practices we extract (Section 3.2). In particular, we look for best practices related to the specification of dependencies between bundles, thus answering our first research question:

Q1 What OSGi best practices are advised?

3.1 Systematic Review Methodology

To perform the identification of best practices related to OSGi dependencies management, we follow the guidelines specified by Kitchenham et al., which include the definition of the research question, search process, study selection, data extraction, and search results [21]. In these regards, **Q1** is selected as the *research question* of the systematic review.

3.1.1 Search process

Given the absence of peer-reviewed research tackling OSGi best practices, we select as primary data sources web resources of the OSGi Alliance and OSGi-endorsed products. The complete list of certified products[5] includes *Knopflerfish*, *ProSyst Software*, *SuperJ Engine*, *Apache Felix*, *Eclipse Equinox*, *Samsung OSGi*, and *KT OSGi Service Platform (KOSP)*. With the aim to identify best practices, we define a *search string* that targets a set of standard *best practices* synonyms, and their corresponding antonyms:

```
((good OR bad OR best) AND (practices OR design)) OR smell
```

Some of the official web pages of the selected organizations provide their own search functionality. However, we seek to minimize the heterogeneous conditions of the searching environment and only use *Google Search* to explore the set of web resources. We use *JSoup*, an HTML parser for Java, to execute the search queries and to scrap the results. We compute all possible keyword combinations from the original search string and execute one query per combination and organization domain. For instance, to search for the `best AND practices` keywords in English-written resources on the OSGi Alliance domain, we define the following Google Search query: `http://www.google.com/search?q=best+practices+site:www.osgi.org&domains:www.osgi.org&hl=en`. We retrieved the resources in January 2018.

[5]https://www.osgi.org/osgi-compliance/osgi-certification/osgi-certified-products/.

3.1.2 Study selection

Figure 1 details the resource selection process we follow in this study. First, we only include web resources written in English in the review. As shown before in the Google Search query, this language restriction is included as a filtering option in all searches: `hl=en`. In the end, the search engine returns a total of 268 resources.[6]

Second, selected documents should describe best practices related to the management of dependencies in OSGi. To this aim, we conduct a two-task selection where we first consider the occurrences of keywords in the candidate resources, and then we perform a manual selection of relevant documents. On the one hand, we count the occurrences of the searched keywords in each web resource (including HTML, XML, PDF, and PPT files). If one of the keywords is missing in the resource, we automatically discard it. Using this criterion, we reduce the set to 156 resources, and finally 87 after removing duplicates.

On the other hand, we manually review the resulting set, looking for documents that address the research question. In particular, if a resource points to another document (through an HTML link) that is not part of the original set of candidates, it is also analysed and, if it is relevant to the study, it is included as part of our data sources. This task is performed by two reviewers to minimize selection bias. In the end, we select 21 web resources to derive the list of best practices related to OSGi dependencies specification. Some of the OSGi-endorsed organizations do not provide relevant information for the study.

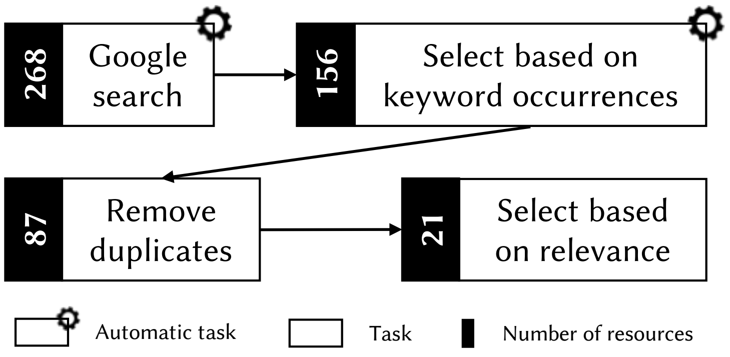

Figure 1: Resources selection of the systematic review.

3.1.3 Data extraction

During the data extraction phase, we consider the organization that owns the resource (e.g., OSGi Alliance), its title, year of publication, authors, and the targeted best practices. To have a common set of best practices, one reviewer reads the selected resources and groups the obtained results in 11 best practices. Afterwards, two reviewers check which best practices are suggested per web resource. Table 2 presents the results of the review. The best practices labels in the table correspond to the best practices presented in Section 3.2.

3.2 Dependencies Specification Best Practices

In this section, we review the best practices identified and summarized in Table 2. We elaborate on the rationale behind each best practice using peer-reviewed research articles and the *OSGi Core Specification Release 6* [37].

[6] `https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts`

Table 2: Systematic review of OSGi dependencies specification best practices.

Resource	Year	Author(s)	Best practices										
			B1	B2	B3	B4	B5	B6	B7	B8	B9	B10	B11
Automatically managing service dependencies in OSGi [27]	2005	M. Offermans	○	○	○	○	○	●	○	○	○	○	○
OSGi best practices! [16]	2007	B.J. Hargrave et al.	●	●	○	●	●	○	●	○	○	○	○
Very important bundles [33]	2009	R. Roelofsen	●	○	○	○	○	●	○	○	○	●	○
OSGi: the best tool in your embedded systems toolbox [15]	2009	B. Hackleman et al.	●	○	○	○	○	○	●	○	○	○	○
bndtools: mostly painless tools for OSGi [4]	2010	N. Bartlett et al.	○	○	●	○	●	○	○	●	○	○	○
Developing OSGi enterprise applications [3]	2010	R. Barci et al.	○	○	○	○	○	○	●	○	○	○	○
Experiences with OSGi in industrial applications [10]	2010	B. Dorninger	○	○	○	○	○	○	●	○	○	○	○
Migration from Java EE application server to server-side OSGi for process management and event handling [20]	2010	G. Kachel et al.	○	○	●	○	○	○	○	○	○	●	○
10 Things to know you are doing OSGi in the wrong way [25]	2011	J. Moliere	●	●	○	○	○	○	●	●	○	○	○
Structuring software systems with OSGi [13]	2011	U. Fildebrandt	●	●	○	○	○	○	●	○	○	○	○
Best practices for (enterprise) OSGi applications [38]	2012	T. Ward	●	●	●	●	○	○	○	●	●	○	○
Building a modular server platform with OSGi [18]	2012	D. Jayakody	●	●	●	○	○	○	●	●	○	○	○
OSGi application best practices [19]	2012	E. Jiang	●	●	○	○	○	○	●	●	●	○	○
TRESOR: the modular cloud - Building a domain specific cloud platform with OSGi [14]	2013	A. Grzesik	○	○	○	○	○	○	●	○	○	○	○
Guidelines [1]	n.d.	OSGi Alliance	○	○	○	○	○	○	●	○	○	○	○
OSGi developer certification - Professional [2]	n.d.	OSGi Alliance	○	○	○	○	○	●	○	○	○	○	○
Using Apache Felix: OSGi best practices [28]	2006	M. Offermans	●	○	○	●	○	○	●	○	○	○	○
OSGi frequently asked questions [11]	2013	Apache Felix	○	○	○	○	○	○	●	○	○	○	●
Dependency manager - Background [12]	2015	Apache Felix	●	○	○	○	○	○	●	○	○	○	○
Best practices for programming Eclipse and OSGi [17]	2006	B.J. Hargrave et al.	●	○	○	○	○	○	○	○	○	○	○
OSGi component programming [39]	2006	T. Watson et al.	●	○	○	○	○	○	●	○	○	○	○

3.2.1 *Prefer package-level dependencies* [B1]

Dependencies should be declared using the `Import-Package` header instead of using the `Require-Bundle` header. The latter creates a tight coupling between the requiring bundle and the required bundle, which is an implicit dependency towards an implementation rather than an interface. Thus, it impacts the flexibility of dependency resolution, as the resolver has only one source to provide the dependency (i.e., the required bundle itself). This also naturally complicates refactoring activities: moving a package from one bundle to the other requires patching all bundles depending on it to point to the new bundle. In contrast, the `Import-Package` header only relies on an interface and various bundles may offer the corresponding package. Finally, `Require-Bundle` automatically imports all the exported packages of the required bundle, which may introduce unnecessary dependencies. This can get worse in some cases, since package shadowing can be introduced unwittingly [37], leading to non-trivial debugging issues.

3.2.2 *Use versions when possible* [B2]

Versions should be set when requiring bundles, or when importing or exporting packages. When a bundle requires another bundle or imports a package, a version range or a minimum required version can be defined. Versions must be consciously used to control the dependencies of a bundle, avoiding the acceptance of new versions that might break the component. Version ranges are preferred over minimum versions, because both upper and lower bounds, as well as all in between versions, are supposed to be tested and considered by bundle developers [9]. In addition, with version ranges the dependency resolver has fewer alternatives to resolve the given requirements, allegedly speeding up the process.

3.2.3 *Export only needed packages* [B3]

Only the packages that may be required by other bundles should be exported. Internal and implementation packages should be kept hidden. Because the set of exported packages forms the public API of a bundle, changes in these packages should be accounted for by the clients [8]. Consequently, the more packages are exported, the more effort is required to maintain and evolve the corresponding API.

3.2.4 *Minimize dependencies* [B4]

Unnecessary dependencies should be avoided, given their known impact on failure-proneness [6] and performance of the resolution process. In the case of OSGi framework and the employment of the `Require-Bundle` header, a required bundle might depend on other bundles. If these transitive dependencies are not considered in the OSGi environment, then the requiring bundle may not be resolved [37]. Moreover, dependencies specification in `Require-Bundle` and `Import-Package` headers may impact performance during the resolution process of the OSGi environment. A bundle is resolved if all its dependencies are available [37]. Presumably, the more dependencies are added to the Manifest file, the longer the framework will take to start and resolve the bundle assuming that all dependencies are included in the environment.

3.2.5 *Import all needed packages* [B5]

All the external packages required by a bundle must be specified in the `Import-Package` header. If this is not the case, a `ClassNotFoundException` may be thrown when there is a reference to a class of an unimported package [37]. This also applies to dynamic dependencies, e.g., classes that are dynamically loaded using the reflective API of Java. The only packages that are automatically available to any bundle are the ones defined in the namespace `java.*`, which are offered by the selected execution environment. However, this environment can offer other packages included in other namespaces. Thus, if these packages are not explicitly imported and the execution environment is modified, they will become unavailable and the bundle will not get resolved.

3.2.6 *Avoid* **`DynamicImport-Package`** [B6]

This header lists a set of packages that may be imported at runtime after the bundle has reached a resolved state. In this case, dependency resolution failures may appear in later stages in the life cycle of the system and are harder to diagnose. This effectively hurts the *fail fast* idiom adopted by the OSGi framework [35]. Also, the **`DynamicImport-Package`** creates an overhead due to the need to dynamically resolve packages every time a dynamic class is used [37].

3.2.7 Separate implementation, API, and OSGi-specific packages [B7]

It is highly recommended to separate API packages from both implementation and OSGi-specific packages. Therefore, many implementation bundles can be provided for a given API, favouring system modularity. The OSGi service registry is offered to select an implementation once a bundle is requiring and using the associated API packages. With this approach, API packages can be easily exported in isolation from implementation packages, allowing a change of implementation if needed. Moreover, implementation changes that result in breaking changes for clients' bundles are avoided. The abovementioned APIs are known as *clean APIs*, i.e., exported packages that do not use OSGi, internal, or implementation packages in a given bundle [37].

3.2.8 Use semantic versioning [B8]

Semantic versioning[7] is a version naming scheme that aims at reducing risks when upgrading dependencies. This goal is achieved by providing concrete rules and conventions to label breaking and non-breaking software changes [32]. Following these rules, a version number should be defined as `major.minor.micro`. In some cases, the version number is extended with one more alphanumerical slot known as `qualifier`. The `major` number is used when incompatible changes are introduced to the system, while the other three components represent backward-compatible changes related to functionality, bugs fixing, and system identification, respectively. The use of semantic versioning supposedly communicates more information and reduces the chance of potential failures.

3.2.9 Avoid splitting packages [B9]

A split package is a package whose content is spread in two or more required bundles [37]. The main pitfalls related to the use of split packages consist on the mandatory use of the **`Require-Bundle`** header, which is labelled as a bad practice, and the following set of drawbacks mentioned in the *OSGi Core Specification* [37]: (i) *completeness*, which means that there is no guarantee to obtain all classes of a split package; (ii) *ordering*, an issue that arises when a class is included in different bundles; (iii) *performance*, an overhead is introduced given the need to search for a class in all bundle providers; and (iv) *mutable exports*, if a requiring bundle *`visibility`*

[7] `http://semver.org/`

directive is set to `reexport`, its value may suddenly change depending on the *`visibility`* value of the required bundle.

3.2.10 Declare dependencies that do not add value to the final user in the `Bundle-Classpath` header [B10]

If a non-OSGi dependency is used to support the internal functionality of a bundle, it should be specified in the `Bundle-Classpath` header. These dependencies are known as *containers* composed by a set of *entries*, which are then grouped under the *resources* namespace. They are resolved when no package or bundle offers the required functionality [37]. Given that a subset of these resources is meant to support private packages functionality, they should be kept as private packages and defined only in the classpath of the bundle.

3.2.11 Import exported API packages [B11]

All the packages that are exported and used by a given bundle should also be imported. This may seem counter-intuitive, as exported packages are locally contained in a bundle and can thus be used without being imported explicitly. Nevertheless, it is a best practice to import these packages explicitly, so that the OSGi framework can select an already-active version of the required package. Be aware that this best practice is only applicable to clean API packages [37].

4 Metrics for OSGi Best Practices

In this section, we introduce a set of OSGi metrics that we specify and use to assess whether OSGi best practices are being followed in a given corpus. Some metrics are defined at the bundle levels and others at the corpus level so that distributions can be easily computed. These metrics are linked to the corresponding best practices and their associated goals, as depicted in Table 3. Note that these metrics rely on an analysis of the Manifest files together with the actual bytecode of the bundles. For instance, we use bytecode analysis to determine whether imported packages and bundles are actually used in the code of a bundle to detect superfluous imports.

We use these metrics to implement our OSGi analysis tool (cf. Section 5) and to report on the use of best practices in the main P2 repository of Eclipse 4.6 (cf. Section 6).

Table 3: Metrics used to assess OSGi best practices.

Level	Best Practice	Description
Corpus	B1	Ratio of bundles using **`Import-Package`** header
	B1	Ratio of bundles using **`Require-Bundle`** header
	B1	Ratio of bundles using both **`Import-Package`** and `Required-Bundle` header
	B3	Number of bundles exporting a given package
	B2	Ratio of bundles with the **`Bundle-Version`** header
	B2	Ratio of bundles with at least a `major.minor` version specification
	B6	Ratio of bundles using the **`DynamicImport-Package`** header
Bundle	B1/B4/B5	Number of imported packages
	B1/B4	Number of required bundles
	B1/B3/B4	Ratio of unused packages imported with the `Required-Bundle` header
	B1/B3/B4	Number of redundant packages. Packages specified in the **`Import-Package`** header and obtained from a required bundle
	B2	Ratio of imported packages with the *`version`* or `specification-version` attribute
	B2	Ratio of exported packages with the *`version`* or `specification-version` attribute
	B2	Ratio of required bundles with the *`bundle-version`* attribute
	B2	Ratio of imported package versions defined as a version range
	B2	Ratio of required bundle versions defined as a version range
	B2	Ratio of imported packages with at least a `major.minor` version specification
	B2	Ratio of exported packages with at least a `major.minor` version specification
	B2	Ratio of required bundles with at least a `major.minor` version specification
	B3	Ratio of exported packages that are never imported
	B1/B4	Ratio of unused packages imported with the **`Import-Package`** header
	B1/B4	Ratio of unused transitive packages imported with the **`Require-Bundle`** header (reexport)
	B1/B4	Ratio of unused transitive packages imported with the **`Import-Package`** header (uses)
	B5	Ratio of used external packages without an explicit import in the **`Require-Bundle`** or **`Import-Package`** headers
	B6	Ratio of imported packages with the `split` directive
	B6	Ratio of exported packages with the `split` directive
	B5	Ratio of imported `java.*` packages
	B1/B5	Number of packages that are both imported and exported within a given bundle

5 An Analysis Tool for OSGi

In this section, we give a brief introduction to the tool we developed for OSGi analysis, recommendation, and refactoring. *We refer the reader to* ***D2.4 – Dependency Inference Components*** *for in-depth information about the tool and how it has been integrated within the CROSSMINER architecture.*

Figure 2 gives an overview of the proposed approach for dependency inference, analysis, and recommendation. As exemplified in Section 6, the tool we developed supports partially or fully all four phases of the process: data extraction, analysis, recommendation, and refactoring.

The analysis tool is fully implemented with Rascal, a meta-programming language that supports source code analysis, transformation, and generation [22]. The main input of the tool is a set of bundles in the form of source code or JAR files that contain Manifest files and Java bytecode. Rascal comes with a built-in Java parser that can analyse either source code or bytecode and store information about method invocations, inheritance hierarchy, field access, etc. in a set of relations named the Java M^3 model [5]. For OSGi Manifest files, we implemented a custom parser that can parse Manifest files according to the syntax defined in the OSGi Specification Release 6 [37].

From a set of JAR files, the tool extracts a Java M^3 model and an OSGi M^3 model. The OSGi M^3 model is an abstraction of the information extracted from the JARs in the form of a set of relations that store actionable information regarding OSGi dependencies. This set of relations forms a dependency graph that can be easily queried by our recommendation and refactoring tools. Besides, any metric provider of the CROSSMINER platform can easily access these relations to define bespoke analysis and recommendations.

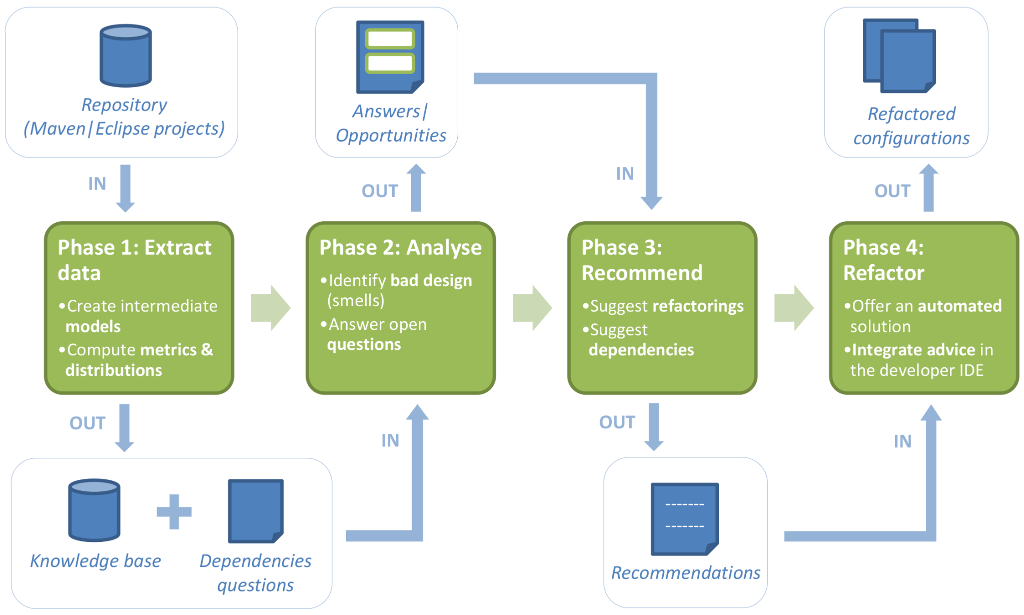

Figure 2: Proposed approach for dependencies inference, analysis, and recommendation.

6 An Analysis of Eclipse P2 Repositories

The best practices we identify in Section 3 emerge from experts of the OSGi ecosystem. The goal of the following two research questions is to assess their relevance and impact critically:

Q2 Are OSGi best practices being followed?

Q3 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle?

Specifically, because beyond their qualitative aspect they are meant to improve performance, we study their impact on the classpath size and resolution time of individual bundles. We first discuss the initial setup and method of our evaluation, and then go through all the selected best practices, aiming at answering our research questions for each of them. After some concluding remarks, we discuss the threats to validity. A complete description of all the artefacts discussed in this section (corpora, transformations, results), along with their source code, is available on the companion webpage.[8]

6.1 Studied Corpus

We use an initial corpus consisting of 1,124 OSGi bundles (cf. Table 4) corresponding to the set of core plug-ins of the Eclipse IDE 4.6 (Neon.1). This corpus was supplied by our use case partner Eclipse Foundation Europe in the context of the CROSSMINER project. The Eclipse IDE consists of a base platform that can be extended and customized through plug-ins that can be remotely installed from so-called *update sites*. Both the base platform and the set of plug-ins are designed around OSGi, which enables this dynamic architecture. The Eclipse IDE relies on its own OSGi-certified implementation of the specification: Eclipse Equinox. Because the Eclipse IDE is a mature and widely-used platform, its bundles are supposed of high quality. As they all contribute to the same system, they are also highly interconnected: the combination of **`Import-Package`**, **`Require-Bundle`**, and **`DynamicImport-Package`** dependencies results in a total of 2,751 dependency links. As a preliminary step, we clean the corpus to eliminate duplicate bundles and bundles that deviate from the very nature of Eclipse plug-ins. This includes:

- *Bundles with multiple versions*. We only retain the most recent version for each bundle to avoid a statistical bias towards bundles which (accidentally) occur multiple times for different versions.
- *Documentation bundles* that neither contain any code nor any dependency towards other bundles are considered as outliers to be ignored. The best practices are specifically about actual code bundles, so these documentation bundles would introduce arbitrary noise.
- *Source bundles* that only contain the source code of another binary bundle are ignored since they are a (technical) accident not pertaining to the best practices either.
- Similarly, *test bundles* which do not provide any functionality to the outside would influence our statistical observations without relating to the studied best practices.

We identify and remove these bundles from the corpus according to their names. The (strong) convention in this Eclipse corpus is that these, respectively, end with a `.doc`, `.source`, or `.tests` suffix. The remaining bundles constitute our control corpus C_0.

[8] https://github.com/msr18-osgi-artifacts/msr18-osgi-artifacts

Table 4: Characteristics of the Eclipse 4.6 OSGi corpus.

Attribute	Value
Initial corpus size	1,124
Number of documentation bundles	17
Number of source bundles	446
Number of test bundles	97
Number of duplicate bundles	192
Studied corpus (C_0)	372
Total size of C_0 (MB)	163.76
Number of dependencies declared in C_0	2,751

6.2 Method

The overall analysis process we follow is depicted in Figure 3 and detailed below.

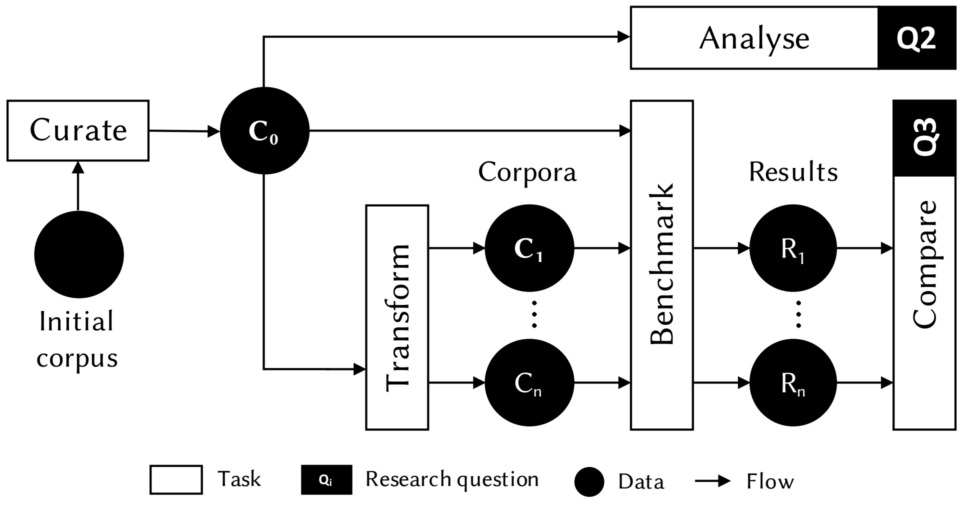

Figure 3: Analysis process.

6.2.1 Selected best practices

For the current analysis, we focus on a subset of the best practices ([B1–B6]) elicited in Section 3, which can be studied using a common research method. The other best practices are interesting as future work: [B7, B10, B11] require distinguishing between implementation and API packages, [B8] requires distinguishing between breaking and non-breaking software changes, and [B9] requires refactoring the source code organization of the bundles in addition to their meta-data.

Are OSGi best practices being followed? (Q2) To answer this research question, we develop an analysis tool, written in Rascal [22], that computes a set of metrics on the control corpus C_0. Specifically, the tool analyses the meta-data (the Manifest files) and bytecode of each bundle to record in which way dependencies and versions are declared, which packages are actually used in the bytecode compared to what is declared in their meta-data, etc. Based on this information, we then count per best practice how many bundles (or bundle dependencies) satisfy it in the corpus. Using descriptive statistics, we then analyse the support for the best practice in the corpus to answer **Q2**. For each best practice, based on the maturity of the Eclipse corpus the hypothesis is that they are being followed ($H2.i$).

Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle? (Q3) To answer this research question, we hypothesize that each best practice would indeed have an observable impact on the size of dynamically computed classpaths ($H3.1.i$) and on the time it takes to resolve and load the bundles ($H3.2.i$). If either hypothesis is true, then there is indeed evidence of observable impact of the best practice of some kind, if not then deeper analysis based on hypothesizing other forms of impact would be motivated. We are also interested to find out if there exists a correlation between classpath size and related resolution time ($H3.3$). Since the latter requires an accurate time measurement setup, while the former can be computed from meta-data, it would come in handy for IDE tool builders (recommendations, smell detectors, and quick fix) if classpath size would be an accurate proxy for bundle resolution time.

Figure 3 depicts how we compare the original corpus C_0 to alternative corpora C_i in which each best practice B_i has been simulated. For each B_i, a specialized transformation $T(B_i)$ takes as input the control corpus C_0 and turns it into a new corpus $C_0 \xrightarrow{T(B_i)} C_i$ where bundles are automatically transformed to satisfy the best practice B_i. For all transformations $T(B_i)$, we ensure that for all bundles that can be resolved in the original corpus, the corresponding bundle in the transformed corpus can also be resolved.

For instance, the transformation $T(B_1)$ transforms every **`Require-Bundle`** header to a set of corresponding **`Import-Package`** headers, according to what is actually used in the bundle's bytecode. Note that bundles using extension points declared by other bundles must use the **`Require-Bundle`** header and therefore cannot be replaced with the corresponding **`Import-Package`** headers. Below, we discuss such detailed considerations with the result of each transformation.

Then, we load every corpus C_i in a bare Equinox OSGi console and compute, for every bundle, (i) the size of its classpath, including the classes defined locally and the classes that are accessible through wiring links according to the semantics of OSGi, and (ii) measure the exact time it takes to resolve it.

Resolution time of a bundle is measured as the delta between the time it enters the `INSTALLED` state ("*The bundle has been successfully installed*") and the time it enters the `RESOLVED` state ("*All Java classes that the bundle needs are available*"), according to the state diagram given in the OSGi specification [37, p. 107].

To report a change in terms of classpath size or performance, we also compute the relative change between observations in C_i and observations in C_0 as $d_{ij} = \frac{v_{0j} - v_{ij}}{v_{0j}} \times 100\%$, where d_{ij} is the relative change between the j^{th} observation of C_0 (i.e., v_{0j}) and the corresponding observation in C_i (i.e., v_{ij}). The median ($\widetilde{x}$) value of the set of relative changes is used as a comparison measure.[9] All performance measurements are conducted on a macOS Sierra version 10.12.6 with an Intel Core i5 processor 2GHz, and 16GB of memory running OSGi version 3.11.3 and JVM version 1.8. Measurements are executed 10 times each after discarding the 2 initial warm-up observations [7].

6.3 Results

To evaluate $H3.3$ we use both scatter plots and correlations (per corpus) that show the relation between our two studied variables, classpath size and resolution time. Figure 4 shows the graph to identify the hypothesized

[9]We use $\widetilde{x_c}$ and $\widetilde{x_p}$, respectively, for classpath size and performance comparisons.

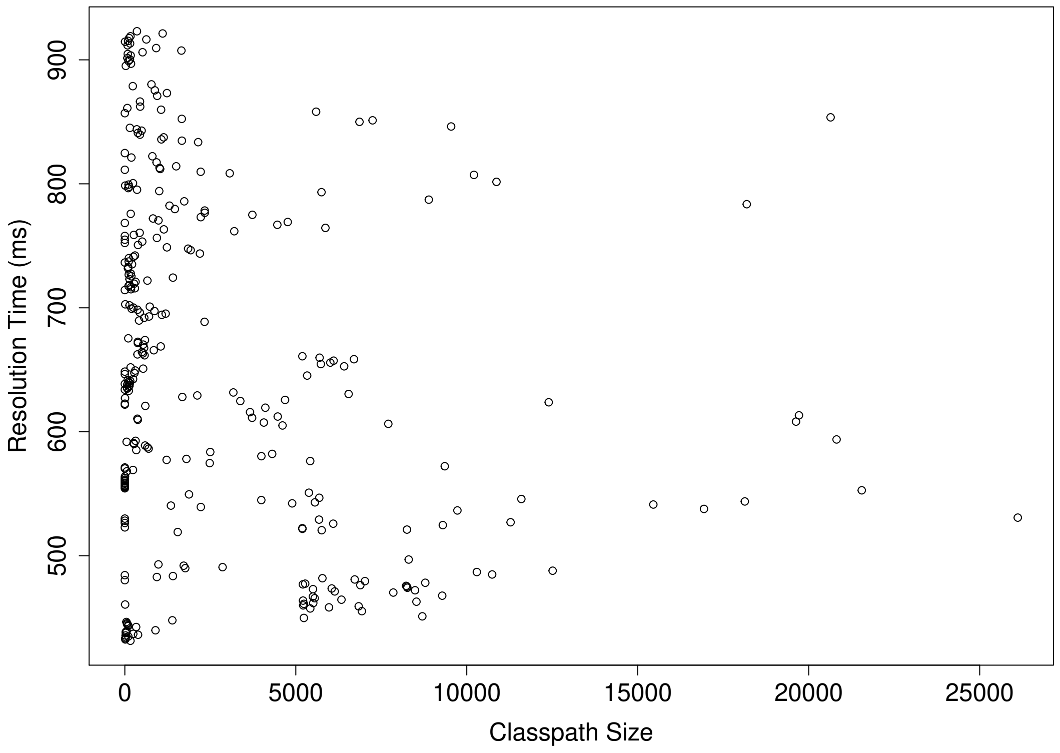

Figure 4: Classpath size is a poor indicator for resolution time (ms) in C_0 ($\rho_0 = -0.17$).

correlation in C_0. Given that there is no linear relation between the variables, we compute the non-parametric Spearman's rank correlation coefficient $\rho_0 = -0.17$, resulting in a weak negative relation. Similar results are observed on all corpora C_i: $\rho_1 = -0.06$, $\rho_2 = -0.17$, $\rho_3 = -0.11$, $\rho_4 = -0.13$, $\rho_5 = -0.17$, and $\rho_6 = -0.17$. According to both visual and statistical analysis, we can reject hypothesis $H3.3$. Therefore, it remains interesting to study these variables independently.

The benchmark results regarding classpath size and resolution time for every corpus C_i, compared to the control corpus C_0, are given in Figures 5 and 6. As reference we show the relation $y = x$ in all figures. Data points positioned over the function show a worse behaviour of C_i with regards to C_0 results, whilst points positioned under the relation elucidates better results. A further description of the obtained results is described in the remainder of the section.

6.3.1 *Prefer package-level dependencies* [B1]

$H2.1$ To test this hypothesis, we count the number of bundles using the **`Require-Bundle`** and **`Import-Package`** headers. We cross-analyse these results by computing the number of extension plug-ins and the number of bundles declaring split packages, which may impact the use of the **`Require-Bundle`** header. The bundles declare 1,283 **`Require-Bundle`** dependencies and 1,459 **`Import-Package`** dependencies. 57.79% of the bundles use the **`Require-Bundle`** header, 50.00% use the **`Import-Package`** header, and 34.95% use both. These results suggest that this best practice tends not to be widely followed by Eclipse plug-ins developers. The declaration of extension points and extension bundles, as well as the use of split packages, contribute to these results. In fact, 38.98% of the bundles in C_0 are extension bundles that require a dependency on the bundle declaring the appropriate extension point to provide the expected functionality. Two of the bundles in C_0 use a `Require-Bundle` dependency to cope with the requirements of split packages.

$H3.1.1$ **and** $H3.2.1$ Transforming bundle-level dependencies to package-level dependencies reduces the classpath size of bundles by $\widetilde{x_c} = 15.40\%$. This is because, in the case of `Require-Bundle`, every exported package in a required bundle is visible to the requiring bundle, whereas the more fine-grained `Import-Package` only imports the packages that are effectively used in the bundle's code. We observe a gain of $\widetilde{x_p} = 7.11\%$ regarding performance (Figure 6).

6.3.2 *Use versions when possible* [B2]

$H2.2$ To tackle this question, we compute the number of versioned `Require-Bundle`, `Import-Package`, and `Export-Package` relations and the proportion of those that specify a version range. 84.80% of the `Require-Bundle` dependencies are versioned, of which 71.97% (i.e., 783) use a version range. In the case of `Import-Package`, 59.22% of the dependencies are versioned, of which 45.14% use a version range. Finally, 24.88% of the 2,620 exported packages tuples are explicitly versioned. The remaining tuples get a value of `0.0.0` according to the OSGi specification. We observe a tendency to use versions when defining `Require-Bundle` relations, which is highly advised given the need to maintain a tight coupling with a specific bundle. Nonetheless, the frequency of version specifications decreases when using `Import-Package` and even more so with `Export-Package`.

$H3.1.2$ **and** $H3.2.2$ The transformation $T(B_2)$ takes all unversioned `Import-Package` and `Require-Bundle` headers in C_0 and assigns a strict version range of the form `[`V, V`]` to them, where V is the highest version number of the bundle or package found in the corpus. In the resulting corpus C_2, we observe that this best practice has no impact on classpath size ($\widetilde{x_c} = 0\%$), and close to zero impact on resolution time ($\widetilde{x_p} = 1.56\%$) of individual bundles.

6.3.3 *Export only needed packages* [B3]

$H2.3$ In this case, we investigate how many of the exported packages in the corpus are imported by other bundles, using either `Import-Package` or `Require-Bundle`, taking versions into account. If an exported package is never imported, this may indicate that this package is an internal or implementation package that should not be exposed to the outside. There may, however, be a fair number of false positives: some of the exported packages may actually be part of a legitimate API but are just not used by other bundles yet. From the whole set of *exported package* tuples, 14.62% are explicitly imported by other bundles. This suggests that a large portion of the packages that are exported are never used by other bundles. Nevertheless, if we also consider packages imported through the `Require-Bundle` header, at least 80.34% of the total tuples are imported by other bundles. The question that arises is: is this situation intended, or is it a collateral effect of the use of `Require-Bundle`?

$H3.1.3$ **and** $H3.2.3$ [B3] has an impact on classpath size in the transformed corpus C_3: exporting only the needed packages results in a $\widetilde{x_c} = 23.27\%$ gain sizewise. We also observe an improvement of $\widetilde{x_p} = 12.83\%$ in terms of resolution time for individual bundles.

6.3.4 *Minimize dependencies* [B4]

$H2.4$ To investigate whether bundles declare unnecessary dependencies, we cross-check the meta-data declared in the Manifest files with bytecode analysis. We deem any package that is required but never used in the bytecode as superfluous. In the corpus, 19.25% of the **Require-Bundle** dependencies are never used locally, i.e., none of the packages of the required bundle are used in the requiring bundle's code. Regarding **Import-Package** dependencies, 13.78% of the explicitly-imported packages are not used in the bytecode. Digging deeper into the relations, we find that the **Require-Bundle** declarations are implicitly importing 15,399 packages that have been exported by the corresponding required bundles. From this set, only 16.50% are actually used in the requiring bundle bytecode. These results suggest that developers tend not to use all the dependencies they declare and that these could be minimized. The situation is much worse in the case of implicitly imported packages through the **Require-Bundle** header, which backs the arguments of [B1].

$H3.1.4$ **and** $H3.2.4$ [B4] has a close to zero impact on classpath size in the transformed corpus C_4 ($\widetilde{x_c} = 0.14\%$). However, the improvement is higher with regards to resolution time ($\widetilde{x_p} = 7.24\%$).

6.3.5 *Import all needed packages* [B5]

$H2.5$ We compute the number of packages that are used in the code but are never explicitly imported in the Manifest file by analysing the bundles meta-data and bytecode. Our analysis identifies 2,194 packages (269 unique) that are never explicitly imported. Overall, 45.70% of the bundles in C_0 use a package that they do not explicitly import (excluding `java.*` packages).

$H3.1.5$ **and** $H3.2.5$ For every package that can be found somewhere in the corpus but is missing in the **Import-Package** list of a given bundle, the transformation $T(B_5)$ creates a new **Import-Package** statement pointing to it. The resulting corpus C_5 does not differ from C_0 in terms of classpath size but appears to be slower in terms of resolution time ($\widetilde{x_p} = -13.35\%$). By creating new explicit dependencies to be resolved, this best practice adds to the dependency resolution process, which in turn may explain this difference.

6.3.6 *Avoid* ***DynamicImport-Package*** [B6]

$H2.6$ In the corpus, only 7 bundles declare **DynamicImport-Package** dependencies, for a total of 9 dynamic relations declared in C_0. 4 of these dynamically imported packages are not exported by any bundle. This may result in runtime exceptions after the resolution of the involved bundles. While there are some occurrences in the corpus of this not-advisable type of dependency, results suggest that developers tend to avoid using the **DynamicImport-Package** header and thus generally follow this best practice.

$H3.1.6$ **and** $H3.2.6$ We do not observe any impact in terms of classpath size, and in terms of performance we observe a gain of $\widetilde{x_p} = 3.47\%$. As our benchmark stops at resolution time and [B6] only has an impact after resolution time, this is unsurprising.

6.4 Analysis of the results

Figure 7 summarizes the overall results regarding relative change of our analysis for classpath size and resolution time. In both graphs we show the relative change of classpath size and resolution time results of C_i with regards to C_0. In the case of classpath size, results obtained above $y = 0$ show the existence of shorter classpaths due to the introduction of the corresponding best practice (Bi). Similarly, in the case of resolution time, results obtained above $y = 0$ entail better performance results when resolving the studied bundles.

6.4.1 Are OSGi best practices being followed? (Q2)

Overall, we observe that most of the best practices we identify are not widely followed in the corpus. This is for instance the case with [B1], despite being the most-widely advocated best practice among the ones we select (cf. Table 2).

Q2: *OSGi best practices related to dependency management are not widely followed within the Eclipse ecosystem.*

6.4.2 Does each OSGi best practice have an observable effect on the relevant qualitative properties of an OSGi bundle? (Q3)

[B1] and [B3] appear to have a positive impact on classpath size (15.40% and 23.27%, respectively), whereas we observe a close to zero impact for [B2], [B4], [B5], and [B6]. Moreover, five of the selected best practices (i.e., [B1], [B2], [B3], [B4], and [B6]) show an improvement on performance that oscillates between 1.55% and 12.83%. [B5] shows a negative impact of 13.35% relative change for the same variable. The absence of larger gains may be explained by the fact that the time required to build the classpath is negligible compared to the other phases involved in bundle resolution (e.g., solving dependencies constraints, as can be observed for [B5]).

Q3: *Only one third of the OSGi best practices we analyse have a positive impact (of up to $\sim$23% change) on the classpath size of individual bundles. Either way, impact on resolution times does not exceed $\pm$13% relative change for all practices.*

6.5 Threats to Validity

Our analysis is naturally subject to threats to validity that challenge our conclusions. Hereafter we present our identified external and internal threats.

6.5.1 External Validity

In principle, the construct of measuring classpath size and resolution time for OSGi bundles can show the presence of a specific kind of impact of a best practice, but not the absence of any other kind of impact. Hence, for where we observed no impact, future analysis of possible other dependent quality factors (e.g. coupling metrics) is duly motivated. However, since the prime goal of OSGi is configuring which bundles to dynamical load into the classpath, any change to OSGi configuration must also be reflected in the classpath. Therefore, in theory, we would not expect any other unforeseen effects when a classpath does not change much.

Although relevant, our research methods did not focus on the downstream effects of OSGi best practices on system architecture or object-oriented design quality in source code. However, minor changes to a classpath may have large impact on those aspects, in particular class visibility may impact software evolution aspects such as design erosion and code cloning. In IDEs specifically, performance is not always a key consideration and other aspects of dependency management remain to be studied as future work.

6.5.2 Internal Validity

With respect to internal validity of the research methods, we calculated classpath size using the OSGi classloader and wiring APIs. Internally, for every bundle, OSGi creates a Java classloader that holds every class local to the bundle, plus all the classes of the bundles it depends on, regardless of the granularity of the dependencies, their visibilities, etc. The OSGi classloaders, on the other hand, *hide* classes from the required bundles when necessary, e.g., when a bundle only requires a few packages from another one using the `Import-Package` header. We aimed to calculate classpath size as seen by the OSGi framework itself, but results may vary if we look at Java classloaders instead. Besides, our analysis and transformation tools may be incorrect in some way. We tried to mitigate this pitfall by having our code written and reviewed by three developers, as well as by writing a set of sanity tests that would catch the most obvious bugs.

The corpus we use for the analysis may also greatly influence the results we obtain for **Q2**—our conclusions only hold for the Eclipse IDE. Nonetheless, we tried to mitigate this effect as much as possible, for instance by taking into account the specificities of the extensions and extension points mechanism within Eclipse which influences our conclusions for [B1]. Similarly, for **Q3**, a different implementation of the OSGi specification may influence the benchmark results.

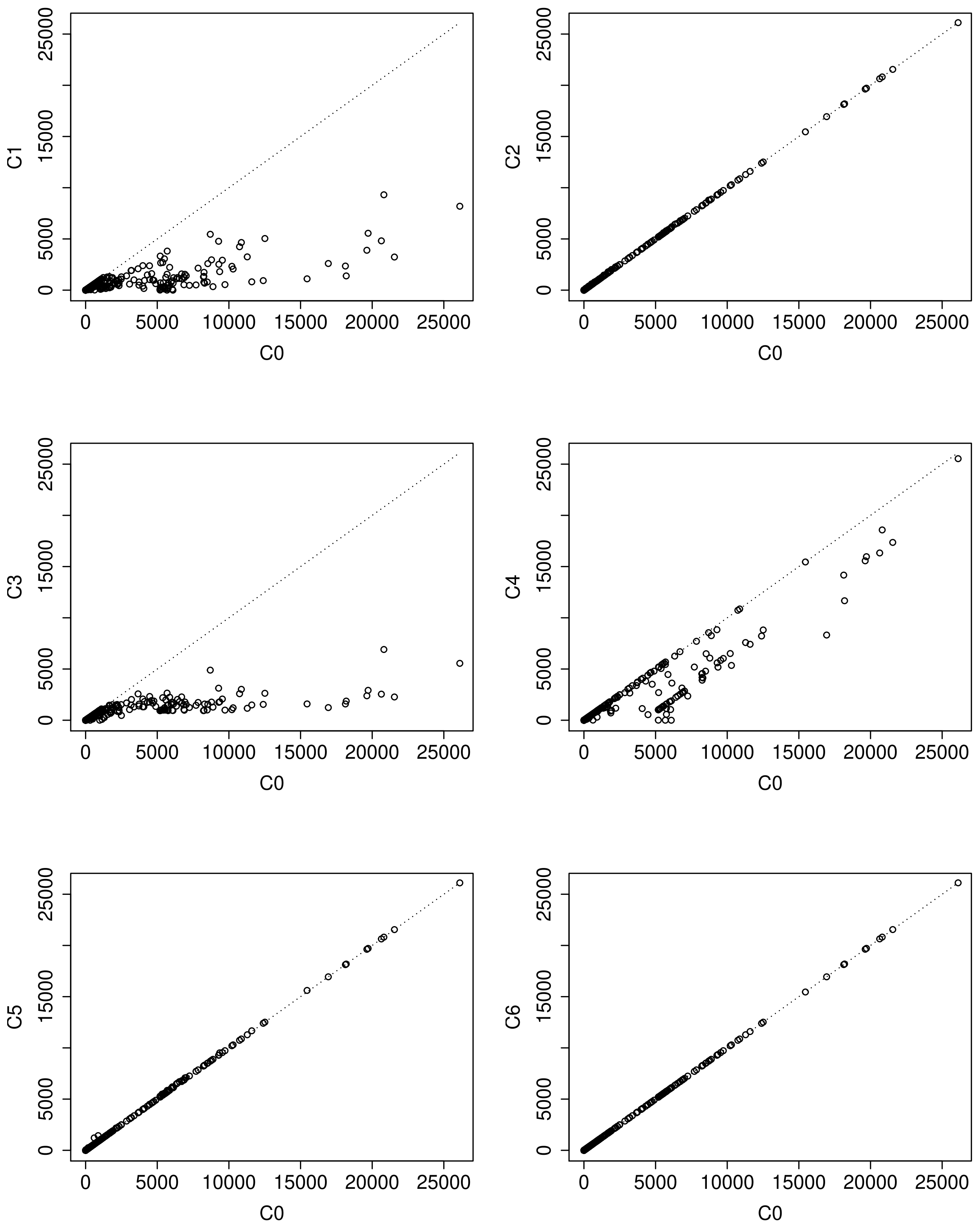

Figure 5: Comparing classpath size of corpora C_i (with best practices B_i applied) to the original corpus C_0.

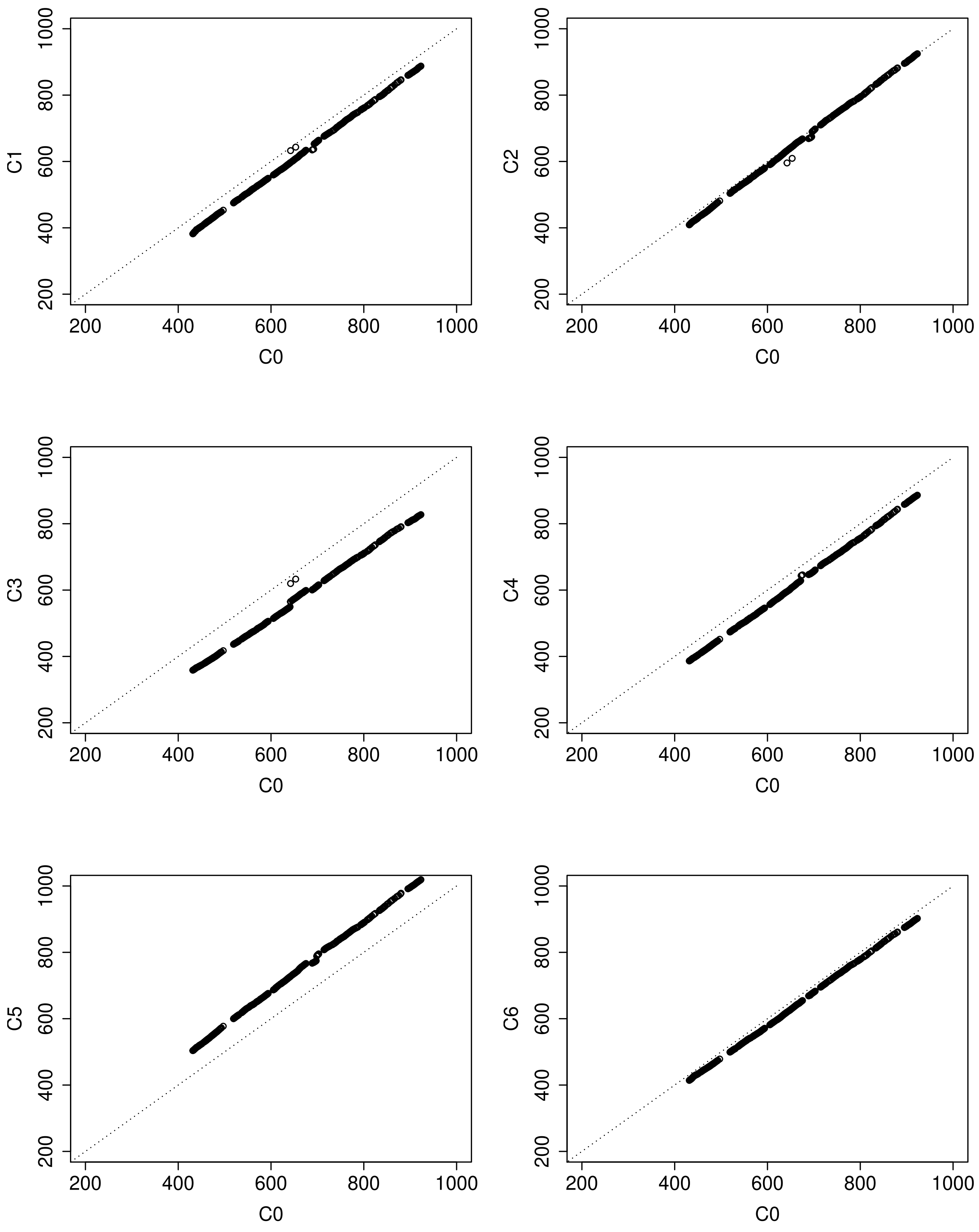

Figure 6: Comparing resolution time (ms) of corpora C_i (with best practices B_i applied) to the original corpus C_0.

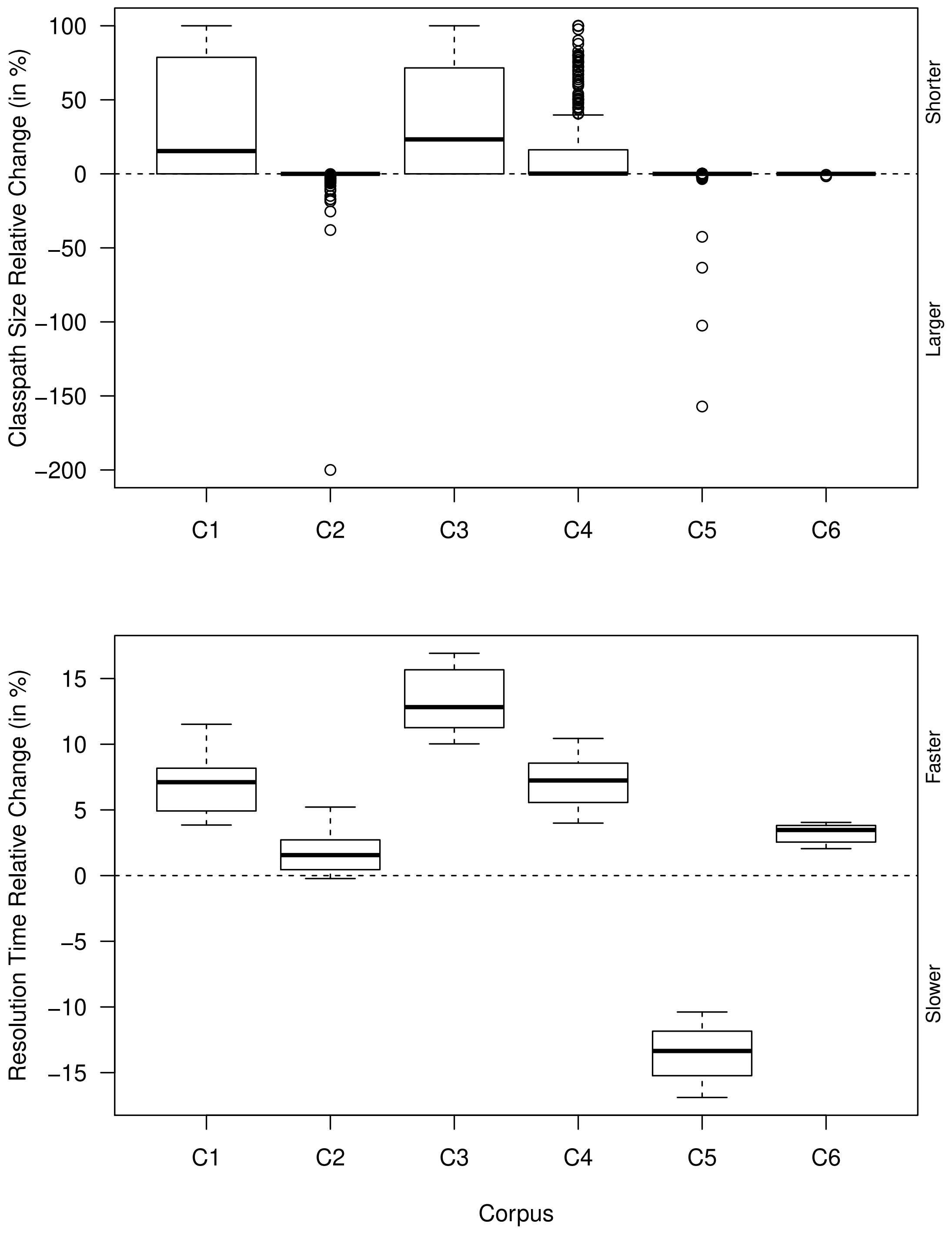

Figure 7: Relative change in classpath size and resolution time between the control (C_0) and transformed corpora (C_i).

7 Results Summary

In this section, we first conducted a systematic review of OSGi best practices to formally document a set of 11 known best practices related to dependency management. We then focused on 6 of them and, using a corpus of OSGi bundles from the Eclipse IDE, we studied whether these best practices are being followed by developers and what their impact is on the classpath size and bundle resolution times. On the one hand, the results show that many best practices tend not to be widely followed in practice. We also observed a positive impact of applying two of the best practices (artificially) to classpath sizes (i.e., [B1] and [B3]), from which we cannot conclude that the respective best practices are irrelevant. Based on this we conjecture most of the identified advice is indeed relevant. Deeper qualitative analysis is required to validate this. On the other hand, the performance results show that OSGi users can expect a performance improvement of up to $\pm 13\%$ when applying certain best practices (e.g., [B3]).

Part II

Mining Dependencies from Apache Maven Projects

8 Introduction

Apache Maven (hereinafter referred to simply as Maven) is a build automation tool highly popular in the Java ecosystem. Maven enables developers to easily declare (i) how to build projects written in Java (source generation, compilation, testing, etc.) and (ii) how to resolve their dependencies. Dependencies are declared in a standard way by the project's developers and are automatically resolved and downloaded from one or many Maven artefact repositories. As of 2018-06-20, 2,881,582 artefacts are hosted on the Maven Central Repository,[10] one of the most popular repository of Maven artefacts against which project dependencies are resolved by default.

In contrast to OSGi, Maven has been the subject of extensive research, in particular regarding the way developers manage their dependencies (e.g., [30, 32, 24, 23, 31]).

In this part:

- We first present some background notions on Maven, POM files, and dependency management in Section 9;
- Then, we detail the approach we follow to extract knowledge regarding dependencies of Maven projects in Section 10.

9 Background: Apache Maven

The central artefact defining how to build and how to resolve dependencies of a Maven project is the Project Object Model (POM), conventionally written down in XML format in a `pom.xml` file. Maven projects may be split in multiple sub-modules, each of which possibly declares its own build process and dependencies in a dedicated `pom.xml` file.

Listing 2 depicts an idiomatic POM file for an imaginary bundle named `Dummy`. It declares the configuration of a project `dummmy`, version `1.0` in the group `org.crossminer.dummy`. The tuple $\langle groupId, artifactId, version\rangle$ is referred to as the unique "coordinates" of the project. Coordinates are used to uniquely identify a Maven artefact; either a local one or one that can be resolved from a remote Maven repository. In particular, coordinates are used to resolve a project's dependencies. In Listing 2, the project `dummy` depends on the `junit` version `3.8.1` in the `junit` group. The <`scope`> attribute specifies that, in this case, the JUnit dependency is only needed at test time. Thus, it will not be included in the compile time and run time classpaths.

Multiple Maven projects can be linked in different ways. A Maven project may include other Maven projects as sub-modules. A Maven project may extend the configuration of a parent Maven project to reuse its build specification.

[10] `https://search.maven.org/`

```
<project>
  <modelVersion>4.0.0</modelVersion>

  <groupId>org.crossminer.dummy</groupId>
  <artifactId>dummy</artifactId>
  <version>1.0</version>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>3.8.1</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
</project>
```

Listing 2: An idiomatic `pom.xml` file.

A popular extension of Maven is Tycho:[11] "*a Maven-centric, manifest-first approach to building Eclipse plug-ins, features, update sites, RCP applications and OSGi bundles*". Specifically, it allows Eclipse developers to automatically build their OSGi-based Eclipse plug-ins which, as stated in Part I, declare their dependencies in a `MANIFEST.MF` file rather than in a `pom.xml` file. Tycho bridges the gap between `MANIFEST.MF` specifications and the Maven build process. Tycho is for instance extensively used in the Eclipse use case of CROSSMINER. The dependency mining, analysis, and refactoring facilities presented in Part I apply in this case.

10 Extracting Dependencies from POM Files

Following the approach we propose for OSGi (cf. Part I), we extract dependencies from Maven meta-data files. To support the inference of Maven project build configuration and dependencies, we built a tool in Rascal that parses all `pom.xml` within a given project (including its sub-modules) to extract the appropriate information. Specifically, the tool dives into the `<dependencies>` section to extract the list of dependencies, their "coordinates" (`groupId`, `artifactId`, and `version`), and their `scope`, i.e., the build phase's classpath it should be included in (compile-time dependencies, run-time dependencies, etc.). An excerpt of the code realizing this task is given in Listing 3. As shown, our tool leverages the built-in XML parser of Rascal to browse through the POM meta-data and extract relevant information related to dependencies.

The resulting dependency model is stored in the Rascal algebraic data type depicted in Listing 4, named the Maven M^3 model. For every POM identified by the `loc id`, it stores a map from logical to physical locations of its (sub-)modules, and exhaustive information about the dependencies it declares, including dependency-specific parameters such as its `scope`. The latter naturally builds a dependency graph between Maven artefacts identified by a unique logical **`loc`**ation. Multiple `MavenModels` can be composed together to build the overall dependency graph of a set of projects, such as the ones analysed by the CROSSMINER platform.

Analysis tools, metrics, recommenders, and refactoring tools can be defined atop the Maven M^3 model. The metric depicted in Listing 5, for instance, uses it to compute the number of optional dependencies declared by a given Maven project analysed by the CROSSMINER platform. It first retrieves the Maven M^3 model of the

[11] `https://www.eclipse.org/tycho/`

```
rel[loc,loc,map[str,str]] getProjectDependencies(loc logical, Node dom) {
 rel[loc,loc,map[str,str]] dependencies = {};
 if(/Node ds:element(_,"dependencies",_) := dom) {
  for(/Node d:element(_,"dependency",_) := ds) {
    groupId = getElementFromDOM("groupId",ds);
    artifactId = getElementFromDOM("artifactId",ds);
    version = getElementFromDOM("version",ds);
    dependencies += {<logical, createProjectLogicalLoc(groupId,artifactId,version),
                  getDependencyParams(d)>};
  }
 }
 return dependencies;
}
```

Listing 3: Extracting project dependencies from POM files in Rascal.

```
data MavenModel = mavenModel (
 loc id,
 rel[loc logical, loc physical, map[str,str] params] locations = {},
 rel[loc project, loc dependency, map[str,str] params] dependencies = {}
);
```

Listing 4: Maven M^3 model in Rascal.

project currently being analysed, and it uses a simple Rascal comprehension to retrieve all the dependencies that have their `optional` attribute set to `true`.

```
@metric{numberOptionalMavenDependencies}
@doc{Retrieves the number of optional Maven dependencies.}
@friendlyName{Number optional Maven dependencies}
@appliesTo{java()}
int numberOptionalMavenDependencies(
 ProjectDelta delta = ProjectDelta::\empty(),
 map[loc, loc] workingCopies = ()) {
 if(repo ←workingCopies) {
  m = getMavenModelFromWorkingCopy(workingCopies[repo]);
  return (0 | it + 1 | <p,d,params> ←m.dependencies, params["optional"]=="true");
 }
 return 0;
}
```

Listing 5: Writing a metric atop the Maven M^3 model.

Part III

Relation to other Work Packages and Requirements Satisfaction

11 Relation to Other Work Packages

This document presents the results we obtained in the context of **Task 2.1** and **Task 2.2**. It builds upon and extends the results presented in **D2.1 – Dependency Inference and Analysis – Initial Report** and **D2.2 – Framework Modelling Components**. More specifically, we report on two predominant frameworks in the Java ecosystem related to dependency management: OSGi (Part I) and Apache Maven (Part II). An in-depth presentation of the software artefacts we developed in this context is available in **D2.4 – Dependency Inference Components**.

There are many natural cross-fertilizations between **WP2 – Mining Source Code** and **WP4 – Mining System Configurations**. While WP2 focuses on the extraction of factual dependencies from project meta-data, source code, and bytecode, WP4 focuses on the analysis of system-level configuration files, which may be used to refine and enrich our own analysis of dependencies. We plan to investigate the relation between dependencies declared at the software level (e.g., OSGi Manifest files and Maven pom files) and at the system level (e.g., Puppet and Docker configuration files). A possible output would be a common formalism for expressing dependency graphs, so that analyses and recommendations can be factorized, when possible.

Similarly, we plan to investigate how the knowledge extracted from dependencies could help identify relationships between projects and libraries, as part of **WP6 – Mining Cross-Project Relationships**. For instance, some of the approaches presented in **D6.3 – The CROSSMINER Knowledge Base – Interim Version** rely on the inference of project dependencies to discover similar projects: the similarity of two projects can be determined by considering the set of common dependencies. More generally, all the metrics and analyses developed in the context of **WP2 – Mining Source Code** are meant to feed the Knowledge Base which acts as a medium between the metrics and the final developer.

The analyses we present in this document are not tailored to a particular project: they are meant to analyse OSGi bundles and Maven projects regardless of the actual client code that may rely on them. A primary goal of CROSSMINER, however, is to tailor the analyses to help developers understand and manage their dependencies directly within their IDE, for the particular project they are currently developing. In the remainder of the project, we will thus strengthen our collaboration with **WP7 – Advanced Integrated Development Environments** to bring our analyses directly in the IDE, through the Knowledge Base, to deliver the appropriate information to the developers when it is needed.

12 Satisfaction of CROSSMINER Requirements

In this section, we present the alignment of the work described in this document with the requirements of CROSSMINER use cases extracted from **D1.1 – Project Requirements**. Specifically, we refer to the requirements listed in *Section 17: Consolidated Requirements and Mapping* for **WP2: Mining Source Code** *related to dependency management*.

Req. No.	Requirement	Priority	Status
D8	Source code mining shall document every metric, possibly with references	SHALL	Full: Every dependency-related metric in the standalone OSGi & Maven analysis project (`https://github.com/crossminer/osgi-analysis-rascal/`). As they will gradually be migrated as metric providers in the CROSSMINER platform, their documentation will follow.
D12	Source code mining shall be able to detect the use of a 3rd-party API function from the source code of a project	SHALL	Full: Our tool leverages Java bytecode analysis to infer precisely which parts of a 3rd-party API are used by dependent projects (cf. D20).
D19	Source code mining shall be able to detect dependencies between libraries/projects (OSGi and Maven) in the form of a dependency graph, and discriminate between test/development/runtime dependencies	SHALL	Full: Our analysis tools can be used to infer both OSGi and Apache Maven dependencies. The extracted dependencies are stored as a set of relations that form a dependency graph and can easily be queried by metric providers. In the case of OSGi, it is not possible to distinguish between development, test, and runtime dependencies in Manifest files; one of the recommended practice in Eclipse is instead to separate tests in dedicated bundles. We can correctly infer the scope of a dependency in the case of Apache Maven.
D20	Source code mining shall be able to infer dependencies from an analysis of the code of the open source framework	SHALL	Full: Reusing and extending previous results from the OSSMETER project, our dependency mining tool automatically infers factual dependencies from the Java bytecode of software projects. This is notably used to support requirement D21.
D21	Source code mining should be able to detect superfluous dependencies and propose a strategy to simplify the dependency graph	SHOULD	Full: Cross-cutting analyses of the handwritten Manifest/POM files and the factual dependencies required in bytecode allow us to detect superfluous dependencies in OSGi bundles. Our tool pinpoints which dependencies are superfluous, letting the developer use this knowledge to refactor her meta-data.

D22	Source code mining may identify over constrained dependencies	MAY	None: In-depth analysis of source code may allow us to warn about overly precise dependencies (e.g., regarding version ranges). This however remains future work, based on previous results for Java analysis.
D23	The components developed within the *Mining Source Code* WP shall interact with the *Mining Natural Language Sources* and *Mining System Configuration* whenever applicable (e.g., when extracting code snippets from plain-text documentation, or to make the configuratioon analysis more accurate based on knowledge extracted from static source code analysis)	SHALL	Partial: While the components we developed do not interact directly with the components from WP3 and WP4 yet, they enable anyone to write metric providers that can query the results of dependency analysis. Doing so, it is possible to write metric providers that cross-fertilize the results of components developed within any WP. The Knowledge Base can later digest the results of different components to provide actionable knowledge to the developers.

Table 5: Satisfaction of CROSSMINER requirements extracted from **D1.1 – Project Requirements**.

References

[1] OSGi Alliance. Guidelines. https://goo.gl/kT4FU6, n.d.

[2] OSGi Alliance. OSGi developer certification - Professional. https://goo.gl/TftHFF, n.d.

[3] Roland Barcia, Tim de Boer, Jeremy Hughes, and Alasdair Nottingham. Developing OSGi enterprise applications. https://goo.gl/5fbom7, 2010.

[4] Neil Bartlett and Peter Kriens. bndtools: mostly painless tools for OSGi. https://goo.gl/FpmhJR, 2010.

[5] B. Basten, M. Hills, P. Klint, D. Landman, A. Shahi, M. J. Steindorfer, and J. J. Vinju. M3: A general model for code analytics in Rascal. In *IEEE 1st International Workshop on Software Analytics*, pages 25–28, 2015.

[6] Veronika Bauer and Lars Heinemann. Understanding API usage to support informed decision making in software maintenance. In *16th European Conference on Software Maintenance and Reengineering*, pages 435–440, Washington, 2012. IEEE.

[7] Stephen M Blackburn, Kathryn S McKinley, Robin Garner, Chris Hoffmann, Asjad M Khan, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. Wake up and smell the coffee: Evaluation methodology for the 21st century. *Communications of the ACM*, 51(8):83–89, 2008.

[8] Bradley E. Cossette and Robert J. Walker. Seeking the ground truth: A retroactive study on the evolution and migration of software libraries. In *20th International Symposium on the Foundations of Software Engineering*, pages 55:1–55:11, New York, 2012. ACM.

[9] A. Decan, T. Mens, and M. Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *24th International Conference on Software Analysis, Evolution and Reengineering*, pages 2–12, Piscataway, 2017. IEEE.

[10] Bernhard Dorninger. Experiences with OSGi in industrial applications. https://goo.gl/hPokNX, 2010.

[11] Apache Felix. OSGi frequently asked questions. https://goo.gl/g2Luqy, 2013.

[12] Apache Felix. Dependency manager - Background. https://goo.gl/758FPJ, 2015.

[13] Ulf Fildebrandt. Structuring software systems with OSGi. https://goo.gl/T9ywmA, 2011.

[14] Alexander Grzesik. TRESOR: the modular cloud - Building a domain specific cloud platform with OSGi. https://goo.gl/dfaV6m, 2013.

[15] Brett Hackleman and James Branigan. OSGi: The best tool in your embedded systems toolbox. https://goo.gl/CDNuWo, 2009.

[16] B. J Hargrave and Peter Kriens. OSGi best practices!, howpublished = https://goo.gl/pfqjev, year = 2007,.

[17] B. J. Hargrave and Jeff McAffer. Best practices for programming Eclipse and OSGi. https://goo.gl/Wp6KfW, 2006.

[18] Dileepa Jayakody. Building a modular server platform with OSGi. https://goo.gl/9tg1ok, 2012.

[19] Emily Jiang. OSGi application best practices. https://goo.gl/qTDxqM, 2012.

[20] Gerd Kachel, Stefan Kachel, and Ksenija Nitsche-Brodnjan. Migration from Java EE application server to server-side OSGi for process management and event handling. https://goo.gl/9zKb2y, 2010.

[21] Barbara Kitchenham, O. Pearl Brereton, David Budgen, Mark Turner, John Bailey, and Stephen Linkman. Systematic literature reviews in software engineering - A systematic literature review. *Inf. Softw. Technol.*, 51(1):7–15, 2009.

[22] Paul Klint, Tijs van der Storm, and Jurgen Vinju. EASY meta-programming with Rascal. In *3rd International Summer School Conference on Generative and Transformational Techniques in Software Engineering III*, pages 222–289, Berlin, Heidelberg, 2011. Springer.

[23] Raula Gaikovina Kula, Daniel M German, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest Maven release. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 520–524. IEEE, 2015.

[24] Shane McIntosh, Bram Adams, and Ahmed E Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, 2012.

[25] Jerome Moliere. 10 Things to know you are doing OSGi in the wrong way. https://goo.gl/TdRh2e, 2011.

[26] Lina Ochoa, Thomas Degueule, and Jurgen Vinju. An empirical evaluation of OSGi dependencies best practices in the Eclipse IDE. In *Proceedings of the 15th International Conference on Mining Software Repositories*, 2018.

[27] Marcel Offermans. Automatically managing service dependencies in OSGi. https://goo.gl/BFT8x2, 2005.

[28] Marcel Offermans. Using Apache Felix: OSGi best practices. https://goo.gl/jmZsYD, 2006.

[29] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Commun. ACM*, 15(12):1053–1058, 1972.

[30] Steven Raemaekers, Arie van Deursen, and Joost Visser. The Maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 221–224. IEEE Press, 2013.

[31] Steven Raemaekers, Arie van Deursen, and Joost Visser. An analysis of dependence on third-party libraries in open source and proprietary systems. In *Sixth International Workshop on Software Quality and Maintainability, SQM*, volume 12, pages 64–67, 2012.

[32] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *14th International Working Conference on Source Code Analysis and Manipulation*, pages 215–224, Washington, 2014. IEEE.

[33] Roman Roelofsen. Very important bundles. https://goo.gl/jznk62, 2009.

[34] Doreen Seider, Andreas Schreiber, Tobias Marquardt, and Marlene Brüggemann. Visualizing modules and dependencies of OSGi-based applications. In *IEEE Working Conference on Software Visualization*, pages 96–100, Piscataway, 2016. IEEE.

[35] Jim Shore. Fail fast [software debugging]. *IEEE Software*, 21(5):21–25, 2004.

[36] Rok Strniša, Peter Sewell, and Matthew Parkinson. The Java module system: Core design and semantic definition. In *22nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, pages 499–514, New York, 2007. ACM.

[37] The OSGi Alliance. OSGi core release 6 specification, Jun 2014.

[38] Tim Ward. Best practices for (enterprise) OSGi applications. `https://goo.gl/wmmNaR`, 2012.

[39] Thomas Watson and Peter Kriens. OSGi component programming. `https://goo.gl/eR5Tmo`, 2006.